

SDEV 1201

Database Fundamentals

LESSON 9

Let's review

Query Syntax

Queries are written with the SELECT statement. The simplified syntax for a select statement is:

```
SELECT [ALL | DISTINCT] column_list
[FROM table_name [, table_name2 [... , table_name]] -- source of data
      [INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN] -- other sources
[WHERE clause] -- filter
[GROUP BY clause] -- aggregate by
[HAVING clause] -- filter by aggregate
[ORDER BY clause] -- sorts results
```

If it looks complex don't worry. All the "clauses" listed don't need to be used when creating a query. However, what clauses you do use MUST be in the order shown above (i.e. HAVING MUST be after GROUP BY for example and not vice versa).

Simple SELECT

```
SELECT  
    FirstName  
, LastName  
, City  
FROM Student;
```

Aggregate Functions

Aggregate functions calculate a single value using the data from many records. If you have used function in Excel such as SUM, AVERAGE, COUNT, etc.... then you have used aggregate functions.

aggregate_function_name ([ALL | DISTINCT] expression)

- *AVG(ColumnName)* returns the average of numeric values. **NULL** is ignored.
- *SUM(ColumnName)* returns the sum of a column containing numeric values.
- *MIN(ColumnName)* and *MAX(ColumnName)* returns the minimum & maximum values from a column of numeric, date, or character values.
- *COUNT(*)* returns the number of records that match the **WHERE** criteria.
- *COUNT(ColumnName)* returns the number of non-null values in a given column for records that match the **WHERE** criteria.

String Functions

- `LENGTH(column | expression)`
- `LEFT(column | expression, length)`
- `RIGHT(column | expression, length)`
- `SUBSTRING(column | expression, start, length)`
- `REVERSE(column | expression)`
- `UPPER(column | expression)`
- `LOWER(column | expression)`
- `LTRIM(column | expression)`
- `RTRIM(column | expression)`

Date Functions

- `Current_Date` returns the current system date
- `Current_Time` returns the current system time
- `DATE_PART(xx, date1)` returns integer representation of the `xx` of `date1`

`xx` represents field for date portions (see next slide).

Date Function Examples

-- You can also use Fields to get the character representation of dates.

`Select To_Char (Current_Date, 'DAY');`

Returns the day of the week (i.e. MONDAY, TUESDAY, etc.). Note that the “Field” is case sensitive: “Day” would return “Monday”.

`Select To_Char (Current_Date, 'MONTH');`

Returns the month name. Note that the “Field” is case sensitive: “Month” would return “September”.

Group By

The **GROUP BY** clause is used with aggregate functions to provide subtotals. We could easily return the average Mark from the grade table. What if we wanted the average Mark for each course? For each student? For each course for each Student? To do this we would use GROUP BY.

```
SELECT StudentID, AVG(Mark) AS AverageMark  
FROM Registration  
GROUP BY StudentID
```

calculates the average mark per Student.

We will usually GROUP BY something unique.

Another way to think of the syntax GROUP BY is the 'for each'.

Having

HAVING is like the **WHERE** clause, except it applies its criteria after **GROUP BY**. For example:

```
SELECT StudentID, AVG(Mark) as AverageMark
FROM Registration
GROUP BY StudentID
WHERE AVG(Mark) > 80
```

Doesn't work. However, Replacing **WHERE** with **HAVING** having does.

```
SELECT StudentID, AVG(Mark) as AverageMark
FROM Registration
GROUP BY StudentID
HAVING AVG(Mark) > 80
```

Joins

For example, if I want to select some fields from the Staff table and other fields from the Position table, which are connected using the PositionID PK/FK

```
select FirstName || ' ' || LastName "Staff Name",  
PositionDescription "Position"  
from Staff  
inner join Position  
on Staff.PositionID = Position.PositionID
```

The INNER JOIN syntax tells the server that we are performing an inner join and identifies that PositionID is the field in the 2 tables that relate them together.

Selecting from 3 or more tables

To select from more than 2 tables the syntax would look like:

```
SELECT field1, field2,... FROM table1  
INNER JOIN table2  
ON table1.joinfield = table2.joinfield  
INNER JOIN table3  
ON table.joinfield = table.joinfield  
INNER JOIN table4  
ON table.joinfield = table.joinfield
```

Outer Joins

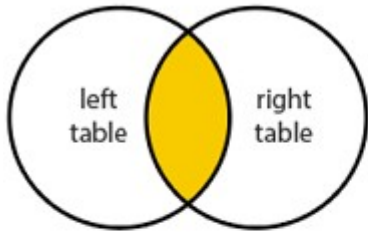
Left Outer Joins and Right Outer Joins both have the same purpose, and you choose which one to use based on the order you list your tables in your select statement.

```
select FirstName || ' ' || LastName "Staff Name", PositionDescription  
"Position"  
from Position  
left outer join Staff  
on Staff.PositionID = Position.PositionID
```

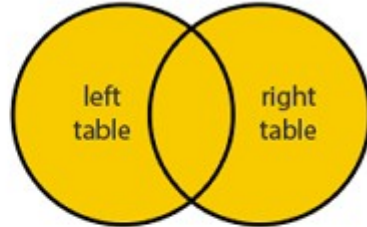
This example states to select ALL the records from the table that is on the LEFT side of the OUTER JOIN statement (Position) and the related records from Staff. The results will include the Assistant Dean position description and a NULL value for staff name.

Types of JOINS

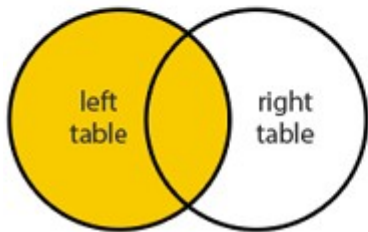
INNER JOIN



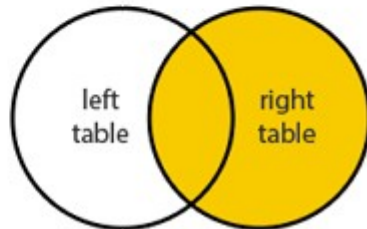
FULL JOIN



LEFT JOIN



RIGHT JOIN



- **INNER JOIN** returns only records that exist in both tables
- **LEFT JOIN** returns all records in **table1**, regardless of whether they exist in **table2**
- **RIGHT JOIN** returns all records in **table2**, regardless of whether they exist in **table1**
- **FULL JOIN** returns all records that exist in either table (and is seldom used)

Union

The union operation lets you combine the data retrieved by multiple SELECT statements.

```
SELECT ...
```

```
UNION [ALL]
```

```
SELECT ...
```

Subqueries

A subquery is a **SELECT** statement inside of another statement.

It's a full and complete **SELECT** statement that could execute on its own.

We often refer to the subquery portion (nested inside the other select) as the inner query and the Select statement around it as the Outer Query.

There are nested and correlated subqueries, and in this course we will focus on nested subqueries only.

ANY, SOME, and ALL operators

ANY or **SOME** compares against any of the values:

```
SELECT StudentID
FROM Registration
WHERE Mark > ANY (SELECT Mark FROM Registration)
-- this returns students with marks higher than at least one other
student
```

ALL compares against all of the values:

```
SELECT StudentID
FROM Registration
WHERE Mark > ALL (SELECT Mark FROM Registration)
-- this returns ... nothing. Why? How can we return just the top grade?
```

ANY, SOME, and ALL operators

In this example we used the subquery as criteria in the having clause.

Breaking it down:

1. The subquery executes first and returns values (1,1,2,1,8,1,1,1) and the outer query is re-written with those values.

Select City

From Student

Group By City

Having Count (StudentID) >= All (1,1,2,1,8,1,1,1);

2. The outer query then executes comparing the count of students in each city to see which count(s) are greater/equal than or equal to all the ones in the subquery. The record from the outer query that is equal to the highest value in the subquery is the city with the most students.

Subqueries Exercise

Today's Objective

Today we will be covering:

- ❑ Correlated Subqueries (Informational only).
- ❑ Cross Joins (Informational only).
- ❑ Views (assessed in SDEV1201).

Theory and reference material for this presentation is based on the “Advanced Select Statement” document in the “Week 4 - Select Statements” section of Brightspace.

Correlated Subqueries

Note: Correlated Subqueries are **not assessed in SDEV1201**.

Correlated subqueries are a bit different than the subqueries we have looked at previously. Those were called nested subqueries. Nested subqueries had the subquery query execute once and then the outer query executed once.

Correlated subqueries work like this:

1. The outer query retrieves a record.
2. The inner query executes using data passed to it from the record retrieved by the outer query.
3. The inner query passes data back to the outer query. This data is used by the outer query to finish processing the record.

The process repeats once for every record processed by the outer query.

Correlated Subqueries

The process repeats once for every record processed by the outer query.

The defining feature of a correlated subquery is that the inner query references (i.e. uses data from) the outer query. Simply put, the inner query “looks” at the outer query.

The following example uses a correlated subquery to find the students whose BalanceOwing is greater than the average payment that student has made.

```
Select Student.StudentID, Student.FirstName, Student.LastName, Student.BalanceOwing
From Student
Where Student.BalanceOwing > (
    Select AVG(Payment.Amount)
    From Payment
    Where Payment.StudentID = Student.StudentID);
```

Correlated Subqueries

Example 2: Find students whose last payment is less than \$500.00

```
Select Student.StudentID, Student.FirstName, Student.LastName,  
Student.BalanceOwing  
From Student  
Where Student.StudentID In (  
    Select Payment.StudentID  
    From Payment  
    Where Payment.Amount < 500 and  
           Payment.StudentID = Student.StudentID);
```


Cross Joins

Note: Cross Joins are **not assessed in SDEV1201**.

The Cross Join produces a Cartesian product of two tables, meaning it returns all possible combinations of rows from the joined tables. This basic syntax is used when you need to combine every row of one table with every row of another table.

Cross Joins are used infrequently but are especially useful for generating datasets for analysis. An administrator could match every student with every club to generate what full club membership would look like.

```
Select Student.StudentID, Student.FirstName, Student.LastName, Club.ClubName  
From Student  
Cross Join Club;
```

In this scenario, there are 16 students and 8 clubs resulting in 128 records returned ($16 * 8 = 128$).

Cross Joins

-- Create a select statement to display every student with every type of payment.

```
Select s.StudentID, s.FirstName, s.LastName, pt.PaymentTypeDescription  
From Student s  
Cross Join PaymentType pt  
Order By s.StudentID, pt.PaymentTypeDescription;
```

In this scenario, there are 16 students and 6 payment types resulting in 96 records returned ($16 * 6 = 96$).

Views

SQL Views

A view is a query that is stored in the database. Views are like a virtual table; it lets us save a query to rerun it easily.

This can help to:

- Simplify data retrieval from complex queries
- Hide the underlying table structure
- Control access to data for different users

SQL Views

Simplifying Data Retrieval

Sometimes we need to code complicated select statements containing joins, aggregates, function, unions, etc. to produce the results we want. A view can simplify this task. Some or all of a complex select statement can be created as a view and future queries of that information can be executed against the view.

SQL Views

Hiding The Underlying Table Structure

By creating a view we create an additional layer between the user and the tables. If the definition of a table changes only the view needs to be altered to accommodate the changes. As long as the view contains the same data as it did before the changes to the underlying table(s) then the user is unaware of the changes and they are of no concern to them.

SQL Views

Controlling Access To Data

By creating a view you can restrict which columns certain users are able to see. Suppose that some information in an employee table should be available to everyone but some columns are restricted information. A view can be created that contains the columns that are available to everyone and permissions can be granted on that view to everyone. Other views could contain other combinations of columns and permissions could be granted accordingly to those views for the appropriate users and groups.

SQL Views

Data Modification

A user can do more than just select data from views. As mentioned before, a view can be treated like a table (with some exceptions). You can update, insert, and delete records in a view just as if you were doing so on an actual table. Any changes to the view are reflected in the records in the table that make up the view.

Here are some points to keep in mind when modifying records through a view in PostgreSQL:

- Modifying records through a view must meet all rules and constraints that were created on the tables that the view is based on.
- You cannot insert, update, or delete from a view that is based on more than one table.
- If the view select statement contains distinct, group by, having, aggregate functions, or includes a union, you cannot modify data with that view.

View Syntax

CREATE [OR REPLACE] VIEW viewname

AS

Select ...

Note: a view cannot reference more than 1600 columns
(depending on column types)

Example

Create View StudentCourseView

As

Select Student.FirstName, Student.LastName, Course.CourseID,
Course.CourseName

From Student

Inner Join Registration

On Student.StudentID = Registration.StudentID

Inner Join Course

On Course.CourseID = Registration.CourseID;

Example

What can you do with this view? Try selecting data from it.

```
Select  FirstName, LastName, CourseName  
From StudentCourseView;
```

Data Modification

We have to follow some rules with views.

- Modifying records through a view must meet all rules/constraints on the table(s) the view is based on.
- An **INSERT** or **UPDATE** from a view cannot affect more than one table in the view.
- You cannot **DELETE** from a view that is based on more than one table.

If the **SELECT** statement contains **GROUP BY**, **DISTINCT**, **TOP**, or **UNION** you cannot modify data in that view.

Example - Insert

Insert into StudentCourseView

(FirstName, LastName, CourseID, CourseName)

Values

('Bob', 'Smith', 'SDEV1201', 'New Database Course');

Fails. You cannot insert data through a view if the view is based on multiple tables.

ERROR: cannot insert into view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

Example - Update

Update StudentCourseView

Set CourseName = 'ChangedCourseName'

Where CourseID = 'DMIT1508';

Fails. You cannot Update through a view if the view is based on more than one table.

ERROR: cannot update view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

Example - Delete

Delete From StudentCourseView

Where CourseID = 'DMIT1508';

Fails. You cannot Delete through a view if the view is based on more than one table.

ERROR: cannot delete from view "studentcourseview"

Views that do not select from a single table or view are not automatically updatable.

Example – Create a one table view

Create View GenericRegistrationInformation

As

Select StudentID, CourseID, Semester

From Registration;

Try to **insert** a student using the StudentCourseView.

Insert into GenericRegistrationInformation

(StudentID, CourseID, Semester)

Values

(200578400, 'DMIT1508', '2005J');

Succeeds. The record is inserted successfully because the view is based on a single table.

Example – One table view

Now attempt to delete that student.

Delete From GenericRegistrationInformation

Where StudentID = 200578400 and

CourseID = 'DMIT1508' and

Semester = '2005J';

Succeeds. The record is deleted successfully because the view is based on a single table.

Altering Or Dropping Views

Views can be altered and dropped with the ALTER and DROP keywords.

SYNTAX:

```
ALTER VIEW viewname  
AS  
Select ...
```

SYNTAX

```
DROP VIEW [IF EXISTS] viewname;
```

Task

Do the “View Exercise” found in the “Week 4” section of Brightspace under Activities.

For reference see “Advanced Select Statement” document in the “Week 4 – Select Statements” section of Brightspace.

Reminder

Quiz on Joins and Select Statements will be next Monday on October 6, 2025.

NOTE: Take-home Coding Assignment Advanced SELECT is available on Brightspace. It is due on Friday, October 10, 2025 at 5:00 PM.